**Azure Data Engineering Project**

**Table of Contents**

**1. Executive Summary \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 3**

**2. Introduction \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 5**

**3. Architecture Overview \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 7**

**4. Environment Setup \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 10**

**5. Data Description & Sources \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 15**

**6. Pipeline Design with Azure Data Factory \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 19**

**7. Databricks Implementation \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 26  
8. Databricks Workflows for the Silver Layer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 32  
9. Gold Layer Implementation (Serving Layer) \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 36  
10. Integration & Reporting Layer \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 40**

**11. Final summary \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 42**

**1. Executive Summary**

This project presents a modern, scalable data engineering architecture implemented using Azure and Databricks, built to efficiently manage data ingestion, transformation, and reporting workflows across a cloud-based data lakehouse platform.

At its core, the architecture follows a Medallion Architecture design pattern, leveraging Azure Data Lake Gen2 and Databricks to organize data into three logical layers: Raw (Bronze), Transformed (Silver), and Serving (Gold).

1. Data Ingestion:
   * Data flows in from two main pipelines:
     + Azure Data Factory (ADF) integrates with GitHub for version-controlled pipeline orchestration.
     + Databricks Autoloader handles incremental data ingestion directly into the Raw Data Store.
   * All incoming data is stored in Data Lake Gen2 (Bronze layer) for traceability and auditability.
2. Data Transformation:
   * Using Apache Spark in Databricks, raw data is cleansed, normalized, and joined across sources to generate structured, analysis-ready datasets stored in the Transformed (Silver) layer.
3. Serving Layer and Output:
   * Business-curated datasets are stored in the Gold Layer (Serving) for consumption.
   * Although Delta Live Tables were not used, data was incrementally loaded from bronze to silver to gold layers, demonstrating practical data flow management.
   * These datasets are integrated with Synapse Analytics for warehousing and Power BI or other reporting tools for visualization and analytics.
4. Key Features:
   * Modular and version-controlled pipelines using GitHub and ADF.
   * Scalable and cost-efficient processing through Databricks Autoloader.
   * Separation of concerns between raw, processed, and curated datasets improves data governance, quality, and maintainability.
   * Enables advanced reporting, data warehousing, and self-service analytics.

This solution supports real-time and batch ingestion, ensuring flexibility in handling diverse data sources. The architecture is robust, secure, and future-ready—optimized for performance, governance, and business insights delivery across enterprise data initiatives.

**2. Introduction**

The field of data engineering is rapidly evolving, and businesses today require scalable, efficient, and modular pipelines to manage ever-increasing volumes of structured and semi-structured data. This project, centered on the Netflix dataset, is designed to simulate a real-world, enterprise-grade ETL (Extract, Transform, Load) process using Microsoft Azure’s modern data engineering tools.

This project does not rely on traditional static workflows; instead, it introduces dynamic and parameterized pipeline development using Azure Data Factory (ADF), Azure Data Lake Storage Gen2, and Azure Databricks. These tools are integrated to form a robust architecture following the Medallion Architecture design — Bronze (raw data), Silver (cleansed and structured), and Gold (analytics-ready). A key aspect of the implementation is that it does not use Delta Live Tables; instead, the data flows from Silver to Gold layer using the same approach adopted in Bronze to Silver, giving full control over transformation logic and data validation without relying on managed table pipelines.

The dataset for this project is derived from publicly available Netflix metadata, including TV shows, movies, categories, cast, directors, and countries. These files are sourced dynamically from a GitHub repository and ingested using HTTP connections and ADF’s Copy Data activity.

By the end of this project, the one will gain hands-on experience with:

* Creating reusable, parameterized pipelines in ADF
* Implementing incremental loads using Databricks AutoLoader
* Writing PySpark transformation logic
* Building tiered data lake layers
* Deploying and managing data in a structured and production-ready manner

This project is ideal for data engineers preparing for interviews or aiming to build a strong portfolio using Azure’s data ecosystem. It also reinforces important cloud concepts such as access control, resource management, and scalable architecture—all of which are vital for real-world data engineering roles.

**3. Architecture Overview**

This project follows the Medallion Architecture pattern, a layered approach that organizes data into structured tiers—Bronze, Silver, and Gold—to progressively refine and enrich the data through the ETL process. The architecture leverages key Azure services like Azure Data Factory (ADF), Azure Data Lake Storage Gen2, and Azure Databricks, with GitHub serving as the external data source.

Key Components and Flow

The architectural design consists of the following stages:

1. Bronze Layer – Raw Data Ingestion

* Source: Data files are hosted on GitHub.
* Ingestion Tool: Azure Data Factory (ADF) uses HTTP connectors to pull the raw data into Azure Data Lake Storage Gen2.
* Storage Zone: Data is stored without modification in the Raw Data Store (Bronze Layer).

1. Silver Layer – Data Transformation

* Tool: Azure Databricks is used to read raw data from the Bronze layer.
* Processing: Data is incrementally loaded using Databricks AutoLoader to ensure efficiency and scalability.
* Transformation: PySpark scripts are used for cleaning, joining, and restructuring the data.
* Output: Cleaned and structured data is stored in the Transformed Data (Silver Layer) container within the same Data Lake Gen2.

1. Gold Layer – Serving Layer

* Purpose: This layer contains analytics-ready data.
* Process: Just like the Bronze to Silver movement, data is moved from Silver to Gold using PySpark transformations within Databricks
* Storage: Final curated datasets are stored in the Serving Layer (Gold) for business consumption.

1. Reporting and Warehousing

* Although not a core implementation in this project, the architecture allows for integration with:
  + Azure Synapse Analytics for data warehousing.
  + Power BI for reporting and dashboards by connecting directly to the Gold Layer.

**Reference Diagram**

The below image illustrates the full data flow from source ingestion to serving:

A diagram of data storage

AI-generated content may be incorrect.

**4. Environment Setup**

Establishing a robust and well-structured cloud environment is the foundational step in building any data engineering project. This section outlines the complete environment setup for the Netflix Azure Data Engineering project using the Microsoft Azure platform.

* 1. Azure Account Setup

To begin with, a free-tier Azure account was created. Microsoft provides a 30-day free trial with $200 in Azure credits, allowing users to explore various services without initial cost. The account can be created by visiting [portal.azure.com](https://portal.azure.com), selecting “Try Azure for Free,” and following the steps to register with a Microsoft account and verify identity via credit card (no charges incurred).

This account grants access to Azure services like:

* Azure Data Factory (ADF)
* Azure Data Lake Storage Gen2
* Azure Databricks
* Azure Synapse Analytics (optional for data warehousing)
* Access connector
  1. Creating Resource Groups

A Resource Group serves as a logical container for managing and grouping Azure services. In this project, a resource group named ResourseG\_NetflixProject was created in the East US region.

A screenshot of a computer

AI-generated content may be incorrect.

This group holds:

* AccessConnector\_netflix – Required for Databricks network integration.
* adf-netflix-asp-ucm – Azure Data Factory instance.
* datalakenetflixasp – Storage account for Data Lake Gen2.
* Netflix-AzureDatabricks-ASP – Azure Databricks workspace.
* Access connector – To access the data from the data lake to the databricks
  1. Data Lake & Container Setup

Using the Azure Storage Account (datalakenetflixasp), a Data Lake Gen2 was provisioned. Hierarchical namespace was enabled to support folder-level structure. Multiple containers were created for different layers of the Medallion architecture:

A screenshot of a computer

AI-generated content may be incorrect.

* raw: Holds unprocessed files ingested from GitHub.
* bronze: Stores files moved by ADF after validation.
* silver: Transformed data generated via Databricks.
* gold: Final analytics-ready output (transformed again using PySpark).
* metastore: Reserved for Unity Catalog or Spark catalog needs.

Each container is private and isolated, ensuring data is secure and permission-controlled.

* 1. Azure Data Factory and Databricks Workspace Creation

A screenshot of a computer

AI-generated content may be incorrect.

* 1. Azure Data Factory (ADF):
  + Service name: adf-netflix-asp-ucm
  + Region: East US
  + Used for: creating dynamic, parameterized pipelines to ingest data from GitHub to Azure Data Lake.
  1. Pipeline components:
* Web Activity (GithubMetadata): Calls GitHub REST API to fetch dataset metadata.
* Set Variable: Captures and stores dynamic values (e.g., filenames).
* Validation (ValidationGithub): Ensures the required files are available in GitHub before loading.
* ForEach Loop (ForAllFiles): Dynamically iterates over the list of filenames, calling the Copy activity.
* Copy Activity (Copy GitHub Data): Pulls each file and stores it in the bronze container.
  1. Azure Databricks Workspace:
  + Name: Netflix-AzureDatabricks-ASP
  + Type: Premium (Trial) to support Unity Catalog and workflows.
  + Region: East US
  + Purpose: To run PySpark code, use AutoLoader for incremental loading, and transition data between layers.

**5. Data Description & Sources**

This project utilizes a rich and relatable dataset from Netflix, which includes a variety of metadata related to movies and TV shows available on the platform. The data is sourced from GitHub and downloaded dynamically through Azure Data Factory (ADF).

Source Repository

The data used in this project is publicly hosted on GitHub:

GitHub Link: [Netflix Dataset on GitHub](https://github.com/JoshuaVemuri/Raw_data_ASP_Project/tree/main/Netflix_Data)

This repository contains multiple CSV files that are used in the ETL pipelines as both fact and lookup tables.

* 1. Main Dataset: netflix\_titles.csv

This is the primary fact table in the project and contains detailed records of Netflix titles including:

* show\_id (unique identifier)
* title
* type (TV Show or Movie)
* director
* cast
* country
* date\_added
* release\_year
* rating
* duration
* genres
* description

Usage:  
This file serves as the central data asset that is pulled into the raw layer via ADF and then incrementally processed and enriched using data from lookup tables.

* 1. Lookup Tables

To normalize the data model and improve analytics, several supporting (lookup) tables are used:

* 1. netflix\_cast.csv
  + Contains show\_id and individual cast members.
  + Helps normalize the cast data into a one-to-many structure.
  + 🔁 Used in joins during the transformation stage in Databricks.
  1. netflix\_directors.csv

Maps each show\_id to one or more directors.

Cleans up and standardizes director names from the main titles file.

2.3 netflix\_countries.csv

Extracts show\_id and associated countries.

Useful for geo-based filtering and aggregation.

2.4 netflix\_category.csv

Maps titles to genres/categories.

Enables genre-based analysis and reports.

These files allow better dimensional modeling by avoiding repetition and enabling efficient lookups and joins in the transformation phase.

* 1. Data Format

All files are in CSV format, which is widely used and supported by both Azure Data Factory and Databricks. During ingestion:

* Delimited by commas (,)
* UTF-8 encoding is assumed
* Handled via parameterized datasets in ADF

These files are loaded dynamically from GitHub using HTTP connectors and moved into the bronze layer of the Data Lake. Validation steps ensure that the files exist before triggering downstream activities.

* 1. Ingestion Strategy

The ingestion process follows this logic:

1. Validation: Check if the target file exists in the GitHub repository.
2. ForEach Loop: Iterate over an array of filenames (cast, directors, countries, category) defined in ADF.
3. Copy Activity: Download each CSV file from GitHub and store it in the bronze container of the Data Lake.

This dynamic and automated strategy eliminates the need for manual file ingestion and supports reuse across environments.

* 1. Benefits of This Structure
* Reduces data redundancy by using lookup tables
* Supports one-to-many relationships (e.g., one show to many actors or countries)
* Enables scalable and extensible data transformation pipelines
* Improves query performance in reporting by reducing joins on wide flat tables

|  |  |  |
| --- | --- | --- |
| Dataset | Rows | Columns |
| netflix\_titles.csv | 6,236 | 9 |
| netflix\_cast.csv | 44,311 | 2 |
| netflix\_directors.csv | 4,852 | 2 |
| netflix\_countries.csv | 7,179 | 2 |
| netflix\_category.csv | 13,670 | 2 |

**6. Pipeline Design with Azure Data Factory**

Azure Data Factory (ADF) is the backbone of data orchestration in this project. It was used to dynamically ingest raw Netflix data from GitHub, validate its presence, and route it to the Azure Data Lake. The design follows best practices for modular, scalable pipelines with dynamic parameters and error handling.

* 1. Creating Linked Services

Linked Services in ADF define connection information to external resources.

1.1 GitHub Linked Service

* Type: HTTP
* Base URL: GitHub repository for Netflix data  
  https://raw.githubusercontent.com/JoshuaVemuri/Raw\_data\_ASP\_Project/main/Netflix\_Data/
* Authentication: Anonymous
* Used to fetch CSV files hosted on GitHub via RESTful API

1.2 Azure Data Lake Linked Service

* Type: Azure Data Lake Storage Gen2
* Authentication: Account Key (or Managed Identity)
* Storage: datalakenetflixasp
* Destination for raw (bronze) data and supports hierarchical namespace

These linked services act as reusable connections across multiple activities in the pipeline.

A screenshot of a computer

AI-generated content may be incorrect.

* 1. Parameterized Dataset Creation

Instead of hardcoding paths, the project uses parameterized datasets to allow dynamic data retrieval and storage.

* 1. HTTP Source Dataset:
  + Parameter: file\_name
  + Constructed relative URL dynamically as:  
    Netflix\_Data/@{dataset().file\_name}.csv
  1. Azure Data Lake Sink Dataset:

Parameters: folder\_name, file\_name

Output path format:  
bronze/@{dataset().folder\_name}/@{dataset().file\_name}.csv

This dynamic approach reduces redundancy and simplifies pipeline maintenance when working with multiple files.

* 1. Copy Activity with Dynamic Pipelines

The Copy Data activity is used to move data from GitHub to the Azure Data Lake.

Key features:

* Source: Parameterized HTTP dataset
* Sink: Parameterized Azure Data Lake dataset
* Enables dynamic loading of multiple files (cast, directors, etc.)
* Timeout and retry policies defined to ensure robustness

This activity is wrapped inside a loop, enabling it to copy several datasets using a single reusable template.

* 1. ForEach Loop, Parameters, and Data Validation

To automate loading multiple files: ForEach Activity

* Accepts an array of dictionaries with folder\_name and file\_name

Array: [

{

"folder\_name" : "netflix\_cast",

"file\_name":"netflix\_cast.csv"

},  
{

"folder\_name" : "netflix\_category",  
"file\_name":"netflix\_category.csv"

},

{

"folder\_name" : "netflix\_countries",  
"file\_name":"netflix\_countries.csv"

},  
{

"folder\_name" : "netflix\_directors",  
"file\_name":"netflix\_directors.csv"

}

]

* Loops through each dictionary and injects values into the Copy Activity dynamically
  1. Validation Step

Before executing the loop, a Validation Activity checks whether a required file (like netflix\_titles.csv) exists in the raw container. This ensures that ingestion happens only when the source is ready.

This mimics real-world checks that prevent pipeline failure due to missing data.

Error Handling and Monitoring Pipelines

Built-In Monitoring

* Monitor Tab in ADF shows run status, success/failure logs, durations, and error messages.
* Helps in debugging and auditing.

Email Alerts (Optional)

* Alerts & Metrics allow setting up notifications on pipeline failure
* Can send email or trigger Logic Apps

Retry and Timeout Settings

* Retry policy configured on Copy Activity
* Default timeout adjusted (e.g., from 7 days to 2 hours)

These settings ensure resilience in case of transient failures, network issues, or data delays

Summary of the ADF Pipeline Flow:

1. Web Activity: Reads metadata from GitHubA screenshot of a computer

   AI-generated content may be incorrect.
2. Set Variable: Stores metadata (optional)
3. Validation Activity: Confirms existence of required file

A screenshot of a computer

AI-generated content may be incorrect.

1. ForEach Loop: Iterates through file list
2. Copy Data Activity: Downloads each file to Data Lake

A screenshot of a computer

AI-generated content may be incorrect.

This shows the successfully run pipeline and the data is copied to the bronze layer

A screenshot of a computer

AI-generated content may be incorrect.

**7. Databricks Implementation:**

This section outlines how Databricks was used for data ingestion, transformation, and management in the Silver and Gold layers, supported by screenshots from your actual notebook implementation.

Unity Catalog Setup

As seen in the sidebar of most screenshots, you used Unity Catalog to organize your data:

* Catalog Used: netflix\_catalog
* Schema: netflix\_catalog.net\_schema
* This enabled structured management of Delta tables across the bronze, silver, and gold layers.

AutoLoader for Bronze Layer Ingestion

leveraged **AutoLoader** to perform incremental ingestion from the raw container in Azure Data Lake.

In the 1\_Autoloader notebook:

df = spark.readStream \

.format("cloudFiles") \

.option("cloudFiles.format", "csv") \

.option("cloudFiles.schemaLocation", checkpoint\_location) \

.load("abfss://raw@datalakenetflixasp.dfs.core.windows.net")

configured checkpointing to track state and avoid redundant processing. I also defined a trigger interval to simulate micro-batch streaming. The data loaded using this stream was written to the Bronze layer.

Parameterized Lookup Table Notebooks

To ensure modularity and reusability, I created parameterized notebooks for Silver-layer processing of lookup tables.

In 3\_Silver\_LookupNotebook, I passed an array of folder names like this:

files = [

{"sourcefolder": "netflix\_directors", "targetfolder": "netflix\_directors"},

{"sourcefolder": "netflix\_cast", "targetfolder": "netflix\_cast"},

...

]

Then in 2\_Silver\_Parameters, I used widgets to dynamically pass values into the notebook:

dbutils.widgets.text("sourcefolder", "netflix\_directors")

dbutils.widgets.text("targetfolder", "netflix\_directors")

This allowed to automate ingestion and transformation of multiple datasets with the same notebook logic.

Silver Layer Transformations

performed extensive transformations to clean and enrich the data. Here’s a breakdown of what I did

Filled Nulls

I used fillna() to assign default values and prevent missing values from affecting analytics:

df = df.fillna({"duration\_minutes": 0, "duration\_seasons": 1})

Converted Data Types

To ensure consistency and accuracy, I explicitly casted fields to correct types:

df = df.withColumn("duration\_minutes", col("duration\_minutes").cast(IntegerType()))

Added Flags & Derived Columns

I created a new column called type\_flag to distinguish between Movies and TV Shows:

df = df.withColumn("type\_flag", when(col("type") == "Movie", 1)

.when(col("type") == "TV Show", 2)

.otherwise(0))

also created ShortTitle and ShortRating using the split() function for more concise labels:

df = df.withColumn("ShortTitle", split(col("title"), ":")[0])

df = df.withColumn("ShortRating", split(col("rating"), "-")[0])

Aggregated & Ranked Records

I applied aggregation logic to get counts by type:

df\_vis = df.groupBy("type").agg(count("\*").alias("total\_count"))

For advanced ranking, I used window functions:

from pyspark.sql.window import Window

from pyspark.sql.functions import dense\_rank

df = df.withColumn("duration\_ranking",

dense\_rank().over(Window.orderBy(col("duration\_minutes").desc())))

Writing to Silver Layer

Once the transformation was complete, I saved the processed data to the Silver layer in Delta format:

df.write.format("delta") \

.mode("overwrite") \

.option("path", f"abfss://silver@datalakenetflixasp.dfs.core.windows.net/{var\_targetfolder}") \

.save()

For lookup tables, I used append mode instead of overwrite.

Automated Multi-File Processing

Using the files array and dbutils.jobs.taskValues.set(), I was able to automate the processing of all lookup datasets. This minimized manual effort and improved consistency.

dbutils.jobs.taskValues.set(key="my\_array", value=files)

ADF Pipeline Validation

Before triggering Databricks notebooks, I validated and copied all raw files using Azure Data Factory.

* built a pipeline with Web activity, Set Variable, Validation, and ForEach loop.
* Each file was validated for presence before being copied.
* This ensured that no notebook was executed on missing data.

Summary

|  |  |
| --- | --- |
| Task | Approach Used |
| |  | | --- | | Raw Data Ingestion |  |  | | --- | |  | | |  | | --- | | AutoLoader with Checkpointing |  |  | | --- | |  | |
| |  | | --- | | Data Validation & Widgets |  |  | | --- | |  | | |  | | --- | | dbutils.widgets, printSchema, fillna() |  |  | | --- | |  | |
| |  | | --- | | Data Type Handling |  |  | | --- | |  | | |  | | --- | | PySpark cast() |  |  | | --- | |  | |
| |  | | --- | | Feature Engineering |  |  | | --- | |  | | |  | | --- | | withColumn(), split(), when() |  |  | | --- | |  | |
| |  | | --- | | Lookup Table Automation |  |  | | --- | |  | | |  | | --- | | Arrays + ForEach + Parameterized Notebooks |  |  | | --- | |  | |
| |  | | --- | | Window Functions |  |  | | --- | |  | | |  | | --- | | dense\_rank() + Window.orderBy() |  |  | | --- | |  | |
| |  | | --- | | Data Persistence |  |  | | --- | |  | | |  | | --- | | Delta format to Silver layer via overwrite/append |  |  | | --- | |  | |
| |  | | --- | | ADF Orchestration |  |  | | --- | |  | | |  | | --- | | ADF Orchestration |  |  | | --- | |  | |

**8. Databricks Workflows for the Silver Layer**

To enable seamless orchestration of data ingestion and transformation for the Silver layer, two Databricks workflows were designed and implemented. These workflows support both lookup datasets and the core Netflix dataset, ensuring consistent, automated, and scalable processing.

1. Workflow: Silver\_notebook

Objective:  
This workflow automates the processing of multiple lookup datasets (e.g., netflix\_directors, netflix\_cast, netflix\_countries, netflix\_category) using parameterized notebooks and a dynamic control structure.

Workflow Components:

* lookup\_locations:  
  Executes the 3\_Silver\_LookupNotebook, which generates an array of JSON objects containing sourcefolder and targetfolder values for each lookup dataset.
* Silver\_notebook\_iteration:  
  Iterates over the output array using a For Each control flow. For each dataset:
  + Data is read from the Bronze layer.
  + Transformations such as type casting, null handling, and formatting are applied.
  + The output is written to the corresponding folder in the Silver layer in Delta format.

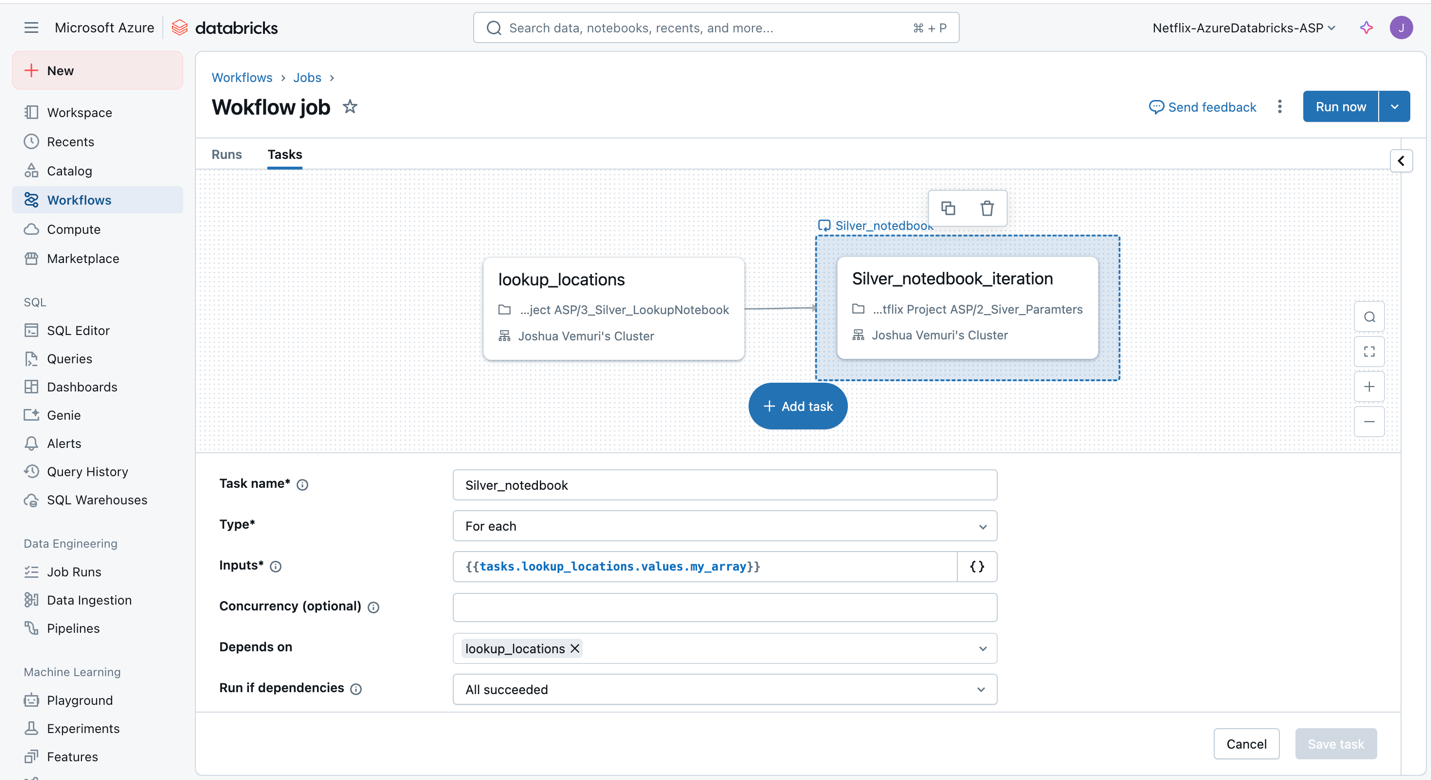
Key Configuration:

* Task inputs are passed using the expression:  
  {{tasks.lookup\_locations.values.my\_array}}
* Parameterization is handled via widgets to support dynamic folder references.

Benefits:

* Reduces redundancy by reusing a single notebook across multiple datasets.
* Promotes scalability through automated iteration over all lookup files.

Visual Reference:



2. Workflow: Week\_Work\_Flow

Objective:  
Controls execution of transformation jobs based on the current day of the week, optimizing compute usage by running heavy jobs only on specific days (e.g., weekends).

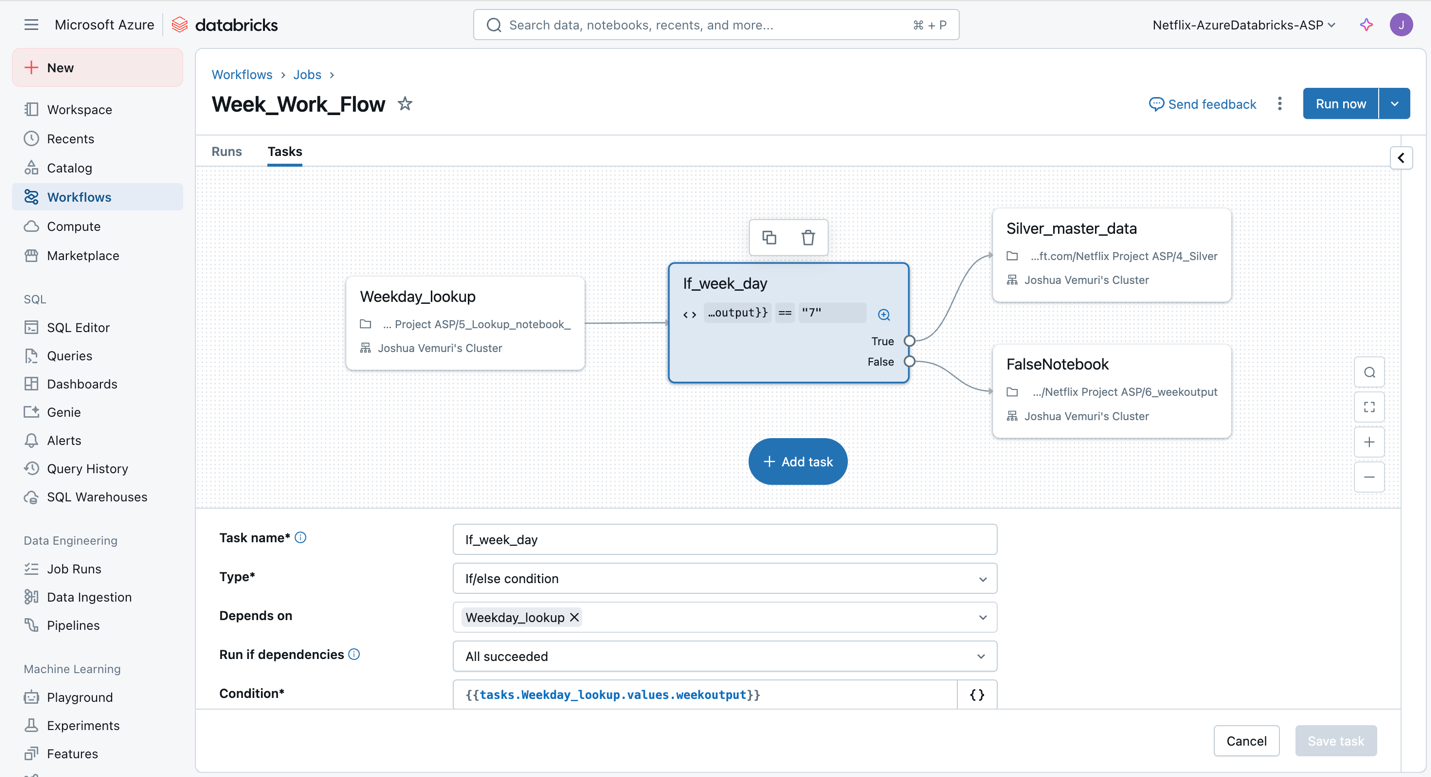
Workflow Components:

* Weekday\_lookup:  
  A notebook task (5\_Lookup\_notebook) that identifies the current weekday and outputs it as a value (1 = Monday, ..., 7 = Sunday).
* If\_week\_day:  
  A decision node that evaluates whether the current day is "7" (Sunday). Based on this condition:
  + If true: Proceeds to Silver\_master\_data.
  + If false: Executes FalseNotebook, a placeholder for alternative logic or skipped execution.
* Silver\_master\_data:  
  Triggers the 4\_Silver notebook, which performs full-scale transformation on the main Netflix dataset. This includes:
  + Filling missing values
  + Type conversion
  + Feature engineering (e.g., ranking, flag columns, short titles)
  + Writing clean data to the Silver layer

Key Features:

* Implements conditional branching using Databricks’ native control flow.
* Enhances resource efficiency by restricting intensive jobs to non-peak days.

Visual Reference:



These workflows form the backbone of the Silver layer data processing strategy, enabling continuous integration of raw and lookup data through robust scheduling, parameterization, and conditional execution. This modular approach ensures both operational efficiency and maintainability as the project scales.

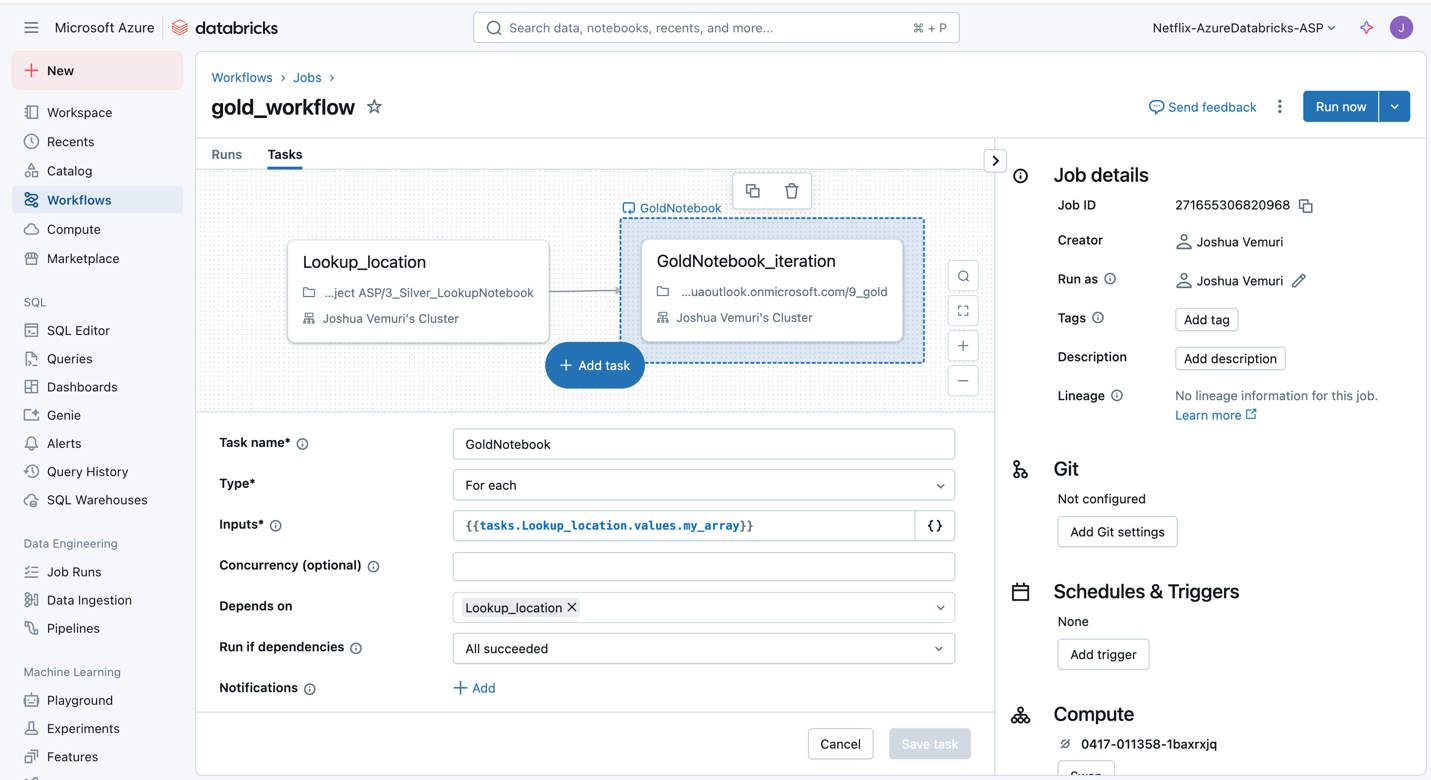
**9. Gold Layer Implementation (Serving Layer)**

The Gold Layer in this project represents the final and most refined version of the data pipeline. This stage focuses on preparing clean, analytical datasets suitable for reporting and decision-making tools.

Workflow Overview: gold\_workflow

The gold\_workflow is designed to iterate over multiple lookup tables and process each through a parameterized notebook. This modular approach ensures scalability and maintainability.

* Task 1: Lookup\_location
  + Executes the 3\_Silver\_LookupNotebook notebook.
  + Returns an array of dictionaries with sourcefolder and targetfolder keys. These values are used to dynamically drive the Gold notebook.
* Task 2: GoldNotebook\_iteration
  + Executes the 9\_gold notebook for each file/folder from the previous task.
  + Utilizes the input array ({{tasks.Lookup\_location.values.my\_array}}) and runs the processing sequentially or in parallel depending on configuration.



Notebook 9\_gold Breakdown

This notebook reads refined silver data and stores it into the gold layer using Delta Lake format. It is fully parameterized, which allows dynamic processing of different datasets.

Step-by-Step Explanation

1. Widget Setup

dbutils.widgets.text("sourcefolder", "netflix\_directors")

dbutils.widgets.text("targetfolder", "netflix\_directors")

Initializes widgets to allow dynamic parameter passing from the workflow.

1. Read Widget Values

var\_sourcefolder = dbutils.widgets.get("sourcefolder")

var\_targetfolder = dbutils.widgets.get("targetfolder")

Retrieves the runtime values passed to the notebook.

1. Read from Silver Layer

df = spark.read.format("delta") \

.option("header", "true") \

.option("inferSchema", "true") \

.load(f"abfss://silver@datalakenetflixasp.dfs.core.windows.net/{var\_sourcefolder}")

df.display()

Reads Delta-formatted data from the Silver container using the specified folder.

Displays the loaded dataframe for validation.

1. Write to the gold layer

df.write.format("delta") \

.mode("append") \

.option("path", f"abfss://gold@datalakenetflixasp.dfs.core.windows.net/{var\_targetfolder}") \

.save()

Saves the refined data into the Gold container using the Delta format in append mode.

**10. Integration & Reporting Layer**

This final layer in the data engineering pipeline enables business users and analysts to derive insights from the curated Gold Layer data using visualization tools. Power BI is used here for interactive reporting and dashboard creation.

Power BI Desktop Integration with Databricks

From the screenshot, the Databricks Marketplace offers seamless Partner Connect integration with Power BI Desktop. This built-in feature allows users to quickly connect Power BI with their Databricks environment, ensuring secure and efficient access to curated datasets for analysis.

Importance of Reporting Integration

1. Real-Time Insights Delivery  
   Data written to Gold Layer (as shown in Databricks notebooks) is structured, cleaned, and often aggregated—making it ready for immediate use in Power BI without further preprocessing.
2. Self-Service Analytics  
   Business users can explore and visualize data independently, reducing reliance on data engineering teams for ad-hoc queries.
3. Unified Data Governance  
   With Unity Catalog and Azure integration, role-based access control ensures secure and compliant access to sensitive data across reporting tools.
4. Ease of Use via Partner Connect  
   As shown in the screenshot, the integration is user-friendly and immediately available through the Databricks Partner Connect, eliminating complex setup steps.

A screenshot of a computer

AI-generated content may be incorrect.

**11. Final summary:**

This project implemented a complete end-to-end data engineering pipeline on Azure, centered around a real-world Netflix dataset hosted on GitHub. The architecture was built using industry-standard cloud services such as:

* Azure Data Factory for data orchestration and ingestion
* Azure Data Lake Storage as the scalable data lakehouse
* Azure Databricks for transformations and layered storage (Bronze, Silver, Gold)
* Power BI for reporting and insights visualization

The process began by pulling raw .csv files from GitHub and ingesting them using Auto Loader with Databricks Structured Streaming. From there, the data was persisted in the Bronze layer for raw storage. It was then cleaned, transformed, and enhanced in the Silver layer using PySpark and workflows. Finally, aggregated and curated data was moved to the Gold layer, optimized for analytics and business consumption.

Modular notebooks, parameterized workflows, and Unity Catalog schemas ensured clean separation of concerns, reuse of logic, and secure, governed access to all datasets.

This project served as a practical demonstration of various advanced data engineering and cloud analytics skills, including:

* **Cloud Integration & Data Ingestion**
  + Use of GitHub as an external data source
  + Auto Loader for incremental loading from cloud storage
  + Dynamic ingestion pipelines via Azure Data Factory
* **Data Lake Architecture (Bronze, Silver, Gold Layers)**
  + Clear separation between raw, cleaned, and aggregated data
* **Data Transformations with PySpark**
  + Null handling, type casting, conditional column creation
  + Aggregations, ranking, and filtering using DataFrame APIs
* **Parameterized Notebooks and Workflows**
  + Use of widgets and Databricks job workflows for automation
  + Iterative processing across multiple folders and files
* **Orchestration with Azure Data Factory & Databricks Workflows**
  + Validation steps, variable setting, ForEach loops in ADF
  + Workflow branching based on weekday conditions for weekly pipelines
* **Integration with Power BI**
  + Enabling seamless visualization via Databricks + Partner Connect